DESIGN AND IMPLEMENTATION OF CONTROL AND MEASUREMENT ELECTRONICS FOR A COMPETITIVE, ULTRA-FUEL-EFFICIENT VEHICLE

A senior project for the Cal Poly Supermileage Vehicle Team.

Bob Somers
rsomers@calpoly.edu

California Polytechnic State University, San Luis Obispo
Computer Engineering Department

Advisor: Dr. Chris Lupo
clupo@calpoly.edu

Table of Contents

Introduction

The Cal Poly Supermileage Vehicle Team builds and races some of the world's most fuel-efficient vehicles. The team had gone dormant in the early 2000s, but was resurrected by a group of Mechanical Engineering students who build a new Supermileage car for their senior project. Since then, their car (the "Black Widow") has been a top-placing competitor at supermileage competitions across the United States, most notably winning the inaugural Shell Eco-marathon Americas in 2007, achieving 1902.7 miles per gallon and taking home the $10,000 grand prize. In 2008 the team placed 2nd at the Eco-marathon Americas with 2752.3 mpg and in 2009 again placed 2nd with 2358.7 mpg [1].

The team is expanding their competitive focus for the 2010 competition by designing and building a new car to enter in the recently-added Urban Concept division while the Black Widow continues to compete in the Prototype division. The Urban Concept vehicle has more practical design constraints, including fully-functioning lighting systems, four wheels instead of three, and restrictions on the driver and luggage compartment dimensions. The team hopes to have the Urban Concept vehicle ready for competition in the 2010 Shell Eco-marathon Americas which takes place March 25-28 at Discovery Green Park in Houston, Texas.

In addition to needing basic electrical systems for the new vehicle, the Supermileage team has expressed interest in developing a custom, state-of-the-art, data acquisition system that can interface with a variety of sensors and log the data for later analysis. The system should be designed specifically for the purposes of collecting and logging data that would be useful to the improvement of ultra-fuel-efficient vehicles such as the Urban Concept vehicle.

To define the scope of the system, the project has been divided into the following five subsystems:

- Basic Electrical Systems
- Instrumentation System
- Data Logging System
- Wireless Monitoring System
- Power Supply System

The requirements and breakdown of each of these subsystems will be described in the background chapter, with an associated priority to the Supermileage team. Because the scope of the project is rather large, the priority is to ensure that in the event that implementation time is running out before the 2010 Eco-marathon, the critical-to-operation systems will be functional and tested.

Background

How the Competition Works

At the Shell Eco-marathon, a competitive run proceeds as follows [2]:

1. The team brings the vehicle to the fuel tent near the starting line, where the gas tank is filled to the fill line marked on the tank.
2. The team wheels the vehicle to the starting line.
3. The vehicle leaves the starting line for its competition run. It completes a set number of laps of the track as required by the rulebook in under the maximum allowed time to ensure a minimum average speed.
4. After crossing the finish line, the vehicle is wheeled back to the fuel tent and the gas tank is refilled to the fuel line. The amount of fuel used to refill the tank is measured and used to extrapolate the fuel economy of the vehicle during the competition run.

Subsystem Breakdown

Here we provide a breakdown of the five major subsystems of the project, as well as information on their requirements and motivation behind their necessity.

Basic Electrical Systems

The basic electrical systems encompass the electronics that are required by the competition rules or for general locomotion. These include:

Engine Control Unit (ECU)

The ECU controls the combustion process in a fuel-injected internal combustion engine. It takes input from sensors such as air and coolant temperature, air pressure in the intake manifold, throttle position, oxygen content in the exhaust, and position of the cylinder to control how much fuel is sprayed into the engine and when the spark plug fires to ignite the fuel. This is a solved problem, and the Supermileage Team is using the commercially-available aftermarket MegaSquirt II ECU to control their Yamaha engine. This falls under a different senior project team which is responsible for delivering a running engine to the Supermileage Team.

Starter

The starter is an electric motor that kick-starts the engine. Responsibility for the motor and wiring falls under the engine project team, however this project will include the electronics that allow the driver to control the starter remotely (from a button on the steering wheel).

Horn

Shell requires all vehicles to be outfitted with a horn that the driver can control [3]. Similarly to the starter, this program will include the electronics to drive the horn remotely from a button on the steering wheel.

Lighting

The Urban Concept rules require vehicles to have two headlights, two front turn indicators, two rear turn indicators, two red brake lights, two rear tail lights, and a starter light [3]. The tail lights and brake lights may be combined as long as they are visibly different when the brakes are being engaged. The starter light alerts other competitors that the vehicle is about to accelerate (since much of the time the engine is off and the vehicle is coasting). The entire lighting system falls under the scope of this project.

Battery

Because the power requirements of this vehicle are complex, the battery is broken into a separate subsystem, the Power Supply System. It is, however, covered in the scope of this project.

The Supermileage Team has indicated that the basic electrical systems have top priority because the Urban Concept vehicle would not be able to compete or pass Shell's technical inspection if they are incomplete.

Instrumentation System

The instrumentation system provides feedback to the driver on the status of the vehicles subsystems as well as other data he needs to make decisions during a competitive run. The following list of important driver information was generated by talking with veteran Supermileage Team driver Tim Lui:

Speed

Both current speed as well as overall average speed are important. Shell requires vehicles to have a minimum average speed for the run to count. Because aerodynamic drag increases proportionally with speed, drivers want to maintain the lowest possible average speed while still staying above the minimum required. By reporting not only instantaneous speed but also average speed the driver can get a sense of how well he's doing.

Engine Data

This includes engine RPM and battery voltage.

Lighting Status

Status indicators for the lights on the vehicle, indicating if the headlights are turned on or off, if the turn signals are on, etc.

Lap Times

As mentioned previously, Shell requires vehicles to maintain a minimum average speed for a competitive run [2]. This is handled by imposing a time limit for vehicles to travel a set distance, in the form of laps around a track, like most race formats. In addition to displaying the average speed, having an accurate timer to track overall run time would be helpful for the driver. This includes an overall timer, as well as a breakdown of times for each lap, with calculations for how many seconds each lap was ahead or behind the optimal time.

The driver will not need to interact with the display of this information very often, but when he does the method of interaction should be simple and compact (i.e. no room for a keyboard and mouse) and should be able to be done with gloves on, since drivers are required to wear a full fire suit (i.e. no capacitive touch screens).

Overall, the job of the instrumentation system is to collect data from a variety of sensors present on the vehicle and present the information in a simple fashion that is custom-tailored to the competitive driving that the vehicle is designed for.

The Supermileage Team has indicated that the instrumentation system is of high priority, as some of the information is critical to the driver being able to do his job effectively.

Data Logging System

The data logging system is primarily responsible for conglomerating all the available sensor data on the vehicle over time and logging it to a permanent storage device. The system should not require any interaction from the driver, and simply silently monitor the state of the vehicle. Sensor data that is available and should be logged includes:

Engine Control Unit

Data provided by the ECU includes the RPM of the engine, the spark advance angle, intake manifold air pressure, air temperature, coolant temperature, throttle position, battery voltage, and calculated airflow through the engine. Other data is available, but not of substantial interest to the team at this time.

Global Positioning System

The GPS system provides latitude and longitude coordinates of the vehicles position, as well as UTC time, altitude above sea level, speed, and compass heading.

Inertial Measurement Unit

The IMU sensor provides instantaneous acceleration on all three axes of the vehicle from a three-axis accelerometer, as well as instantaneous rotational velocity on all three axes from two gyroscopes.

Lighting System State

This includes status of the head lights, turn signals, and brake lights. Also, by monitoring the brake lights, the system can implicitly log when and for how long the vehicle was braking.

The data logging system should provide an easy method of retrieving the data from the vehicle after a run is completed, and preferably export the data in a standard format for later analysis.

The Supermileage Team has indicated that the data logging system has a medium priority, because while it would be nice to collect data during competition for later analysis, it is not essential to the operation of the vehicle.

Wireless Monitoring System

The wireless monitoring system can best be described as a live visual data logger. The system would allow all the sensor data to be broadcast wirelessly to a laptop while the driver is focusing on his competitive run. Since the information can be displayed on a large computer screen, this could allow for a much richer display of information with tools to dive into the data as the vehicle is competing.

Effectively this would function as an expanded instrumentation system, but without omitting certain information. For example, the driver has little need to see the position from the GPS sensor, but the remote monitoring software could display the position readings as breadcrumb trails to track the vehicles progress along the track.

The Supermileage Team has indicated that the wireless monitoring system has low priority and should be the first subsystem to be cut if implementation time becomes short.

Power Supply System

The power requirements of this vehicle are rather complex. From a high level perspective, the car requires a battery that provides power to all the electrical systems, but given that there are a fair number of sensors and embedded electronics onboard (which use a variety of voltage levels for their power rails) it's more complicated than simply connecting up a battery.

Most of the vehicle electronics such as the starter motor, ECU, and ignition system, and fuel injector run off of 12V. However, some of these devices sink a large amount of current. For example, the starter motor sinks nearly 21 Amps during cranking [4]. The interfacing of the control electronics (such as the button on the steering wheel) which operate at TTL-level voltages and currents with the high-power systems is especially important for the long-term health and reliability of the electrical system.

There are mainly two gateways of concern here. Firstly, power coming into the electronics from the battery (which provides standard automotive 12V DC) must be conditioned and stepped down to 3.3V and 5V for the low-level electronics. It should also be fused to protect the low current electronics from possible power surges from the battery.

Secondly, there are two parts where control from the low-voltage electronics interfaces with the high-current 12V devices: the starter motor and the horn. In both cases, care should be taken to ensure that the controls are sufficient electrically isolated from the high-power devices.

The Supermileage Team has indicated that the Power Supply System should have a high priority, since it is critical for operation of the vehicle.

System Description

While the overall system is broken into five major subsystems, these systems overlap and integrate with each other very tightly. Because the physical components of the system may play a role in one or more of the five subsystems, it's easier to describe how specific components in the overall design are connected and what their responsibilities are. Essentially, instead of describing the system from the top down, this section will describe its operation from the bottom up.

Lighting System

Two different technologies were compared for the lighting system: traditional incandescent bulbs and super-bright LEDs. Several factors were considered, ranked by importance:

1. Power consumption
2. Weight
3. Lifespan
4. Visibility
5. Cost

For the most important three factors, super-bright LEDs were far and away the winner. They use drastically less power than traditional incandescent lights because far less energy is wasted through heat dissipation, they weigh less than bulbs, and the lifespan is much longer.

In terms of visibility, they were essentially even, but Shell doesn't have any specific requirements on how visible the lights must be, so it even if the LEDs are slightly less visible it wasn't enough of a concern to prevent LED technology from winning out.

The cost of building custom super-bright LED light modules is significantly more than the immediate cost of incandescent bulbs, but the cost of replacing bulbs over time would add up, and would prove to be more of a reliability hassle than spending the extra money to go with LEDs.

Critical Design Decisions

Several critical design decisions were made to ensure that the LED lighting system would integrate well with the Urban Concept vehicle:

Darlington Driver for High Current Sink

The light modules are driven by a single microcontroller output, but most microcontrollers can only source or sink rough 40 mA max from their I/O pins [4]. Instead, the light modules use a Darlington Driver, which is essentially two cascaded Bipolar Junction Transistors to allow the module to sink large amounts of current [6]. With a 4x6 array of LEDs each sinking 20 mA, the overall current draw of each module is nearly half an Amp.

Instead of powering the entire array of LEDs, the microcontroller toggles the base of the first BJT, which then sinks current for the entire array. Since the LEDs don't need to be individually controlled, current limiting resistors aren't needed for each individual LED.

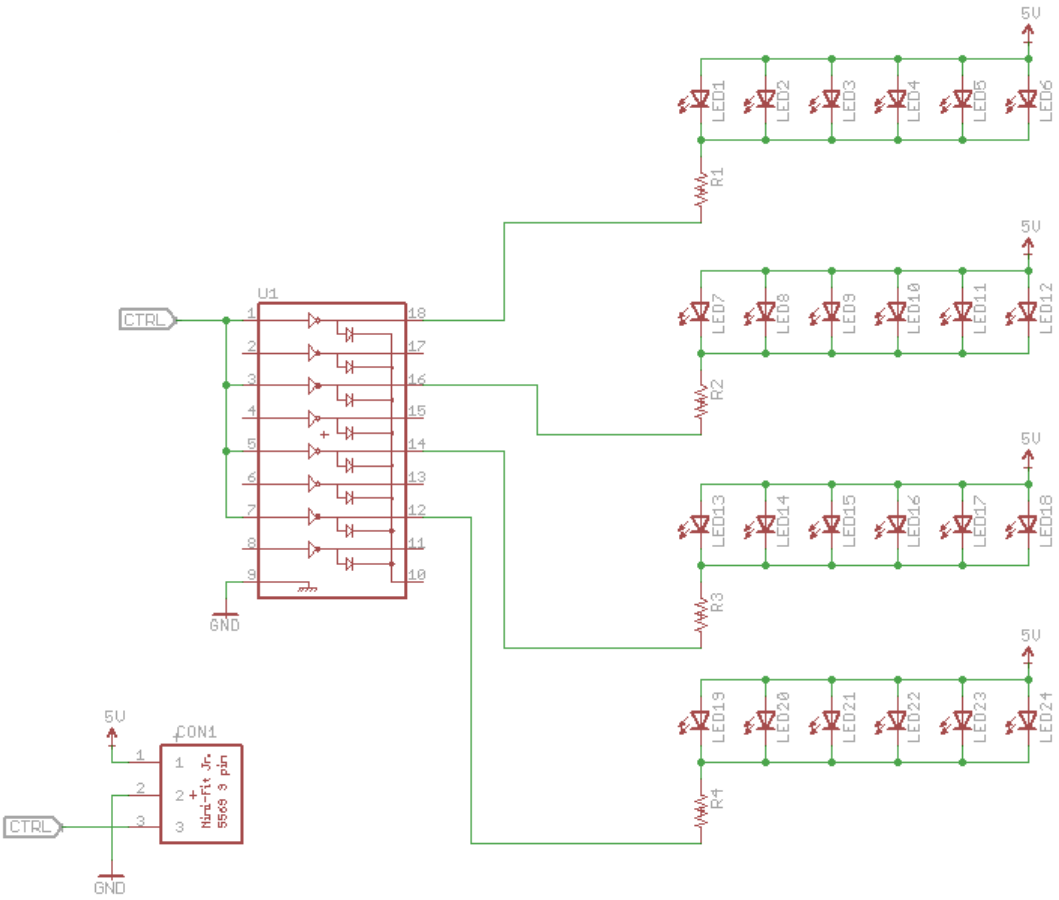
Common PCB Layout

The printed circuit board fabrication was graciously donated by Advanced Circuits. To help reduce their tooling costs, a common board design was made for all the lights. When assembling the modules, one only has to choose the correct color LEDs and associated current-limiting resistors to get a different colored light.

Molex Mini-Fit Jr. Connectors

The lights will be permanently installed inside the fairing, which is the flexible exterior shell of the vehicle. The fairing can be removed when the team needs access to the inside of the vehicle, so it's important that the lights and be quickly and easily connected and disconnected from the controlling electronics, which are mounted to the chassis of the vehicle. This connection needs to be quick and reliable. Molex Mini-Fit Jr. connectors were chosen for this, because they're a very common connector type (used in all standard ATX motherboards) and provide quick, reliable connections and disconnections and have a high current rating on each pin.

Overall the design of the light modules themselves is pretty simple.



The Darlington array takes in the control signal from the microcontroller and grounds its outputs its turned on. The grid of 24 LEDs is split across 4 channels of the 8-channel Darlington. A single ½ watt resistor is used for each channel to limit the overall current for the channel. Lastly, a 3-pin Molex Mini-Fit Jr. connector is used to connect power, ground, and control for the light module.

The resistors values should be chosen based on the forward voltage of the LEDs to limit the current in each channel to 20 mA. For example, if the forward voltage of the LEDs used is 2V, the resistor value can be found using Ohm's Law as follows:

$$I_{LED} = 20 \text{ mA}$$
$$I_{Array} = 6 \cdot I_{LED} = 120 \text{ mA}$$

$$V_{LED} = 2\text{V}$$
$$V_{Darlington} = 1\text{V}$$
$$V_{Total} = V_{LED} + V_R + V_{Darlington}$$

$$5\text{V} = 2\text{V} + V_R + 1\text{V}$$
$$V_R = 2\text{V}$$

$$V_R = I_{Array} \cdot R$$
$$R = \frac{V_R}{I_{Array}} = \frac{2\text{V}}{120\text{ mA}}$$
$$R = 16.67\ \Omega$$

The following table shows the part numbers and forward voltages for the LEDs used for the light modules, as well as the calculated ideal resistance values and chose real-world resistors.

Color	Digi-Key Part No.	Forward Voltage	Ideal Resistance	Actual Resistor
White	160-1728-5-ND	3.3 V	5.833 Ω	6.2 Ω
Amber	C503B-AAN-CY0B0251-ND	2.1 V	15.833 Ω	16 Ω
Red	C503B-RAN-CY0B0AA1-ND	2.1 V	15.833 Ω	16 Ω

Sensors

The following lists all the components of the system, and what subsystems they play a role in.

Venus 634FLPx GPS Module

The Venus GPS module sold by SparkFun Electronics is a small breakout board for the incredibly powerful Venus GPS chip. It provides an exceptionally high refresh rate (10 Hz) for consumer chips, and guarantees closer than 2.5m accuracy. It also can go from power on to satellite lock (a "cold start") in just 29 seconds and draws only 28 mA of current during tracking. It outputs GPS data in the standard NMEA format over a 3.3V UART with a configurable baud rate [7].

Because the GPS sensor provides position, altitude, heading, and speed, it plays an important role in the instrumentation system.

6 DOF "Razor" IMU

The Razor IMU provides an inexpensive way to measure 6 degrees of freedom (acceleration and rotational velocity along all three axes). It provides one 3-axis accelerometer and 2 gyros which provide ±3g acceleration and ±300°/sec range. It's compact in size and consumes very little power [8] [9] [10] [11].

Because the system only provides analog outputs, an analog-to-digital converter (ADC) is required to sample the voltages into useable digital numbers. For this purpose, an Atmel ATMega168 was used as a bridge, since it provides an onboard ADC with 10 bits of precision and an onboard UART for communication. The ATMega168 was clocked at 7.372 MHz, since that's plenty fast enough for ADC sampling and it provides a perfect clock divider for UART communications at any standard baud rate.

FTDI FT4232H Mini Module

While not strictly a sensor per se, the FT4232 Mini Module provides access to four 3.3V UARTs over USB [12]. Because of this, it plays a critical role in sensor communication with the host PC.

Dashboard

The system uses a "virtual dashboard" approach to allow for maximum flexibility and expandability. Rather than a traditional dashboard with dials and indicators, a virtual dashboard is displayed on a 7" touch screen in front of the driver. This allows controls and indicators to be moved around as the team sees fit, and future controls can be added with minimal effort.

Because the driver may need to occasionally interact with the dashboard, the selection was limited to resistive screens so that it could be operated with gloves on.

The following components play a pivotal role in the dashboard:

Via EPIA PX10000G Pico-ITX Computer

The PX10000G was chosen because it offers a complete computer environment, is extremely compact, and consumes very little power. It's explicitly designed for use in embedded systems. It provides a basis for communicating with the sensors on the system, logging that data to permanent storage, and displaying a visual GUI to the driver [13]. Because of this, it plays a role in the instrumentation system, the data logging system, and the wireless broadcast system. A first attempt was made to have the PX10000G run Linux, but poor driver support ultimately required the system to run a stripped down version of Windows XP. To ensure plenty of free disk space was still available on the 8GB flash disk, a free tool called nLite was used to strip all unnecessary components out of the Windows installation [14].

Dynamix 710 Touch Screen

The touch screen is used to display the virtual dashboard to the driver, as well as allow for minimal interaction. It takes a standard VGA display signal and has an industry standard VESA mounting bracket making fabrication of a custom bracket for mounting very straightforward. It operates at standard automotive 12V and draws only 950 mA of current [15].

Qt Cross-Platform GUI Toolkit

Qt is an open-source cross-platform GUI toolkit written in C++ and maintained by Nokia. It provides a very high quality framework for quickly and easily writing portable GUI applications, in addition to some platform-agnostic abstractions of things like network and file system devices and a thorough standard library [16]. It was originally chosen so that development could be done on a Windows machine and deployed on the Linux computer, but since it ended up ultimately running Windows this ended up being of lesser importance. The toolkit still provides an excellent interface for quickly developing and modifying GUI applications.

Power Supply

Because the power requirements of the electronics system were so complex (12V for the touch screen, 3.3V for sensors, and 5V for the lighting system) an automotive ATX power supply was chosen to provide power.

The M4-ATX Automotive DC-DC Power Supply is extremely tolerant of out-of-range voltages, providing clean power with as little as 6V input or as much as 30V. Its highly efficient design produces very little heat and it provides up to 250 watts of power over a standard ATX power connector [17]. The ATX standard has voltage rails for 12V, 5V, and 3.3V.

For the power supply to power more than just the embedded computer, the appropriate voltage rails had to be broken out. On the mainboard (described in more detail below), these voltages are broken out to the necessary components as well as transferred to the embedded PC. Each power rail was checked to ensure there was sufficient headroom so that no device would be pulling too much power from a particular rail.

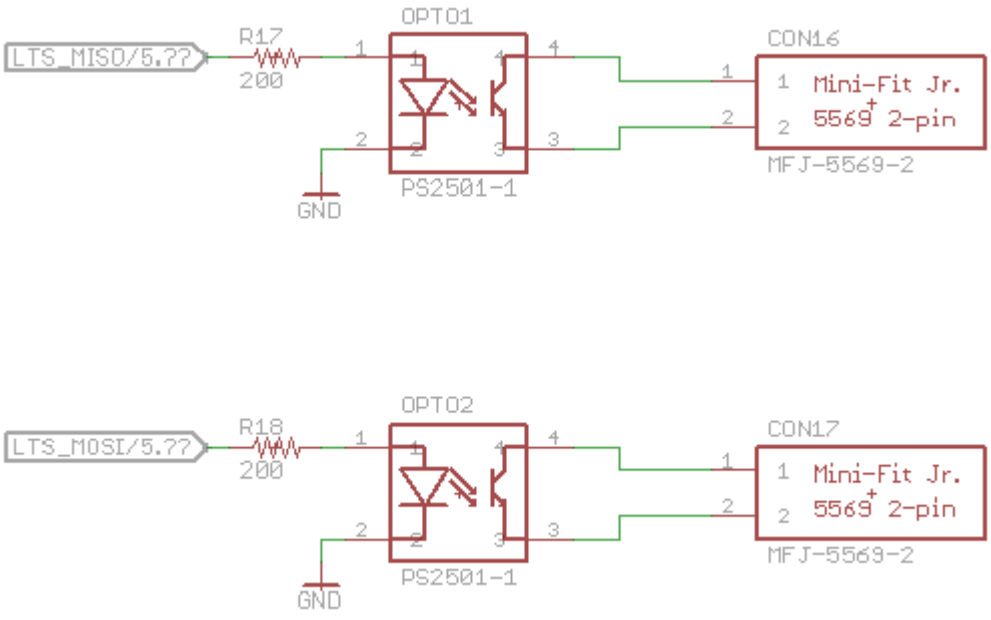
The power supply itself is also fused with a standard automotive “Mini-ATO” style fuse rated at 25 Amps.

Mainboard

The mainboard serves as both a physical mounting substrate and an electrical routing component for the entire system. The power supply and embedded computer are both mounted directly to the mainboard, but it also houses its own components such as the GPS and IMU sensors, wireless radio, and microcontrollers for controlling the lighting systems and converting analog IMU measurements to digital numbers.

Care had to be taken when routing the PCB to ensure that power traces were of sufficient width, since the 3.3V, 5V, and 12V rails could have significant power draw depending on the devices connected to them. To allow for additional expansion of the system, 2 connectors for each power rail were broken out to exterior connectors.

The mainboard also houses two connectors which are used to drive the starter and the horn (high power devices) with low voltage electronics. To do this, a connector is broken out which allows connection to a 12V automotive relay. However, ensure electrical isolation, an optoisolator is used on the mainboard to drive the relay.



The optoisolator uses an LED and a phototransistor internally to optically connect two different circuits. When the microcontroller applies power to its I/O pin, it turns the LED on, which triggers the

phototransistor to switch on the other side of the circuit [18]. The high power side connects a relay to ground when it is switched on, and that relay switches the high power device (starter motor or horn). This way, the control electronics are completely electrically isolated from the high power device they are controlling.

Evaluation

The evaluation of the completed system was somewhat difficult, due to time and resource restrictions facing the team. Approximately two weeks before competition, the decision was made by the team lead that substantial parts of the Urban Concept vehicle would not be ready in time, and that it was better to ensure they were properly engineered than rush the car to compete in the 2010 Eco-marathon Americas. The electronics system (all parts covered by this senior project), however, was not one of the elements holding up the vehicle. The project was complete.

Due to the actual vehicle being unavailable for real-world testing, substantial accommodations were made to ensure that real-world testing could be done on the electronics system before it is finally installed in the vehicle.

The following areas were evaluated to ensure the system would function properly upon installation:

Wireless Transmission Range

One of the areas of most concern was the transmission range of the XBee radio modules. This was primarily a concern because the budget was altered after the design review to use a less powerful (and less expensive) module than previously planned to keep the cost for the system under $1,500. Although the datasheet claims up to a 6 mile range with “high gain antennas” on the less powerful modules [19], it never specifies what antenna or environmental conditions they used to obtain this number. Because the antenna and environment can make a huge impact on RF performance, it was deemed necessary to range-test the radios to determine what kind of performance the team could reasonably expect.

Luckily, Digi International (the manufacturer of the modules) provides a free software utility called X-CTU which not only is useful in configuring the modules, but also range-testing them [20]. With the assistance of a fellow engineer, I headed out to Foothill and Los Osos Valley Road intersection with the modules, antennas, and two laptops to get a relatively unobstructed, flat, test area.

The minimum specifications of this test required the modules to maintain reliable communication with little to no packet loss over a distance of at least 0.1 mile. This was determined because the track radius at the Houston competition site is approximately 0.1 mile. For future competitions, (for example, if they

move the competition back to the speedway in Fontana, CA) this metric may not be adequate and the radios may need to be range tested again.

Since the X-CTU software provides a running percentage of the number of dropped or corrupted packets, we ran the test for 30 seconds each time and generated three percentage ranges for what we considered reliable, manageable, and unacceptable communications performance.

Reliability Classification	Percentage of Packets Correctly Delivered
Reliable	≥ 90%
Manageable	89% - 41%
Unacceptable	≤ 40%

After concluding the tests, I determined the modules had "reliable" performance within the 0.1 mile metric. They started to fade into the "manageable" classification nearing the 0.2 mile mark, and crossed into "unacceptable" territory at the quarter-mile mark. Thus, the radios were deemed suitable for competition in Houston, however, their performance may need to be re-evaluated for future competition sites.

System Integration Testing

Since the system requires significant interaction between its many parts, system integration testing was very important to ensure it would not only function correctly, but be reliable during competition.

All of the subsystems were connected together and extra connectors and wiring harnesses were ordered to ensure that a test setup could be constructed. The steering wheel inputs, lighting systems, sensors, and data logging components were all connected and booted up to simulate their final installation in the vehicle.

In addition to simulating a run of the vehicle by starting the dashboard and watching it respond to stimuli from the sensors, the system was left collecting data and communicating with sensors overnight as an endurance test to ensure that there were no long-running issues such as memory leaks or communications breakdowns that could be detrimental to reliability. The system passed with flying colors, and was deemed ready for final installation in the vehicle.

Conclusions

This project provided me with an excellent experience to hone my engineering skills and work on a large deliverable over the course of 23 weeks. It was an excellent culmination to my Computer Engineering education because it included design and implementation from almost every level of the hardware and software stacks, from power supply and PCB design up through microcontrollers and firmware and finally up through GUI presentation code on top of the operating system. After completing this project, I feel not only like I understand the theory behind everything in the stack, but can implement any part of it as well.

There are many directions this project could head in the future. Once the team has more experience using the system for data acquisition and live performance monitoring, they'll probably have a better idea of what specific features, if any, they would like to improve or tweak. The benefit to the original design of the system is that it is enormously flexible in all areas. The dashboard can be reconfigured simply by modifying the software and recompiling it. Additional hardware and measurement equipment can be added by plugging it into the exposed USB or Ethernet ports, and they can draw reliable, clean, protected power from the expansion power jacks built into the rear of the PCB at several useful logic-level voltages.

The main improvement that can be made in the future is developing analysis tools to help derive meaning from the collected data. Since very little processing is done on the values recorded (for the sake of data integrity), some areas of data will need post-processing to make more sense out of the measurements. For example, the IMU is subject to a certain amount of noise, which can be filtered out by software using something like a particle filter.

In addition, some software that is able to "play back" a simulation of a recorded run from a data file would be very valuable to the team as well. This would allow the engineers and drivers to replay successful and unsuccessful runs, notice differences, and help isolate mechanical problems or driving patterns that are detrimental to fuel economy.

References

[1] Cal Poly Supermileage Team Blog. http://cpsmv.blogspot.com.

[2] Shell Eco-marathon Americas 2010 Rulebook: Chaper II. http://www-static.shell.com/static/ecomarathon/downloads/2010/americas_chapter_2_rules_2010.pdf.

[3] Shell Eco-marathon Ameicas 2010 Rulebook: Chapter I. http://www-static.shell.com/static/ecomarathon/downloads/2010/2010_SEM_rules.pdf.

[4] Yamaha C3 Scooter 2007-2009 Repair Manual. http://servicemanuals.ecrater.com/product.php?pid=4373771.

[5] Atmel AVR ATmega168 Datasheet. http://www.atmel.com/dyn/resources/prod_documents/8271.pdf.

[6] Toshiba ULN2803APG Darlington Driver Array Datasheet. http://www.toshiba.com/taec/components2/Datasheet_Sync/393/22738.pdf.

[7] Skytraq Venus 634FLPx Datasheet. http://www.sparkfun.com/datasheets/GPS/Modules/Skytraq-Venus634FLPx_DS_v051.pdf.

[8] 6DOF Razor IMU Schematic. http://www.sparkfun.com/datasheets/Sensors/IMU/6DOF-Razor-v11.pdf.

[9] STMicroelectronics LPR530AL Dual Axis Pitch and Roll Analog Gyroscope Datasheet. http://www.sparkfun.com/datasheets/Sensors/IMU/lpr530al.pdf.

[10] STMicroelectronics LY530ALH High Performance Analog Gyroscope Datasheet. http://www.sparkfun.com/datasheets/Sensors/IMU/LY530ALH.pdf.

[11] Analog Devices ADXL335 3-Axis Accelerometer Datasheet. http://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf.

[12] FTDI FT4232H Mini Module Datasheet. http://ftdichip.com/Documents/DataSheets/Modules/DS_FT4232H_Mini_Module.pdf.

[13] VIA PX10000G Pico-ITX Embedded PC User's Manual. http://www.via.com.tw/servlet/downloadSvl?id=472&download_file_id=5801.

[14] nLite Windows Installation Pre-packager. http://www.nliteos.com.

[15] Dynamix 710 Touch Screen Monitor. http://dynamixcomputers.com/dynamix_710_touch_screen_monitor.html.

[16] Qt – A Cross-platform Application and UI Framework. http://qt.nokia.com.

[17] M4-ATX 6-30V Intelligent ATX Power Supply Installation Guide. http://resources.mini-box.com/online/PWR-M4-ATX/PWR-M4-ATX-manual.pdf.

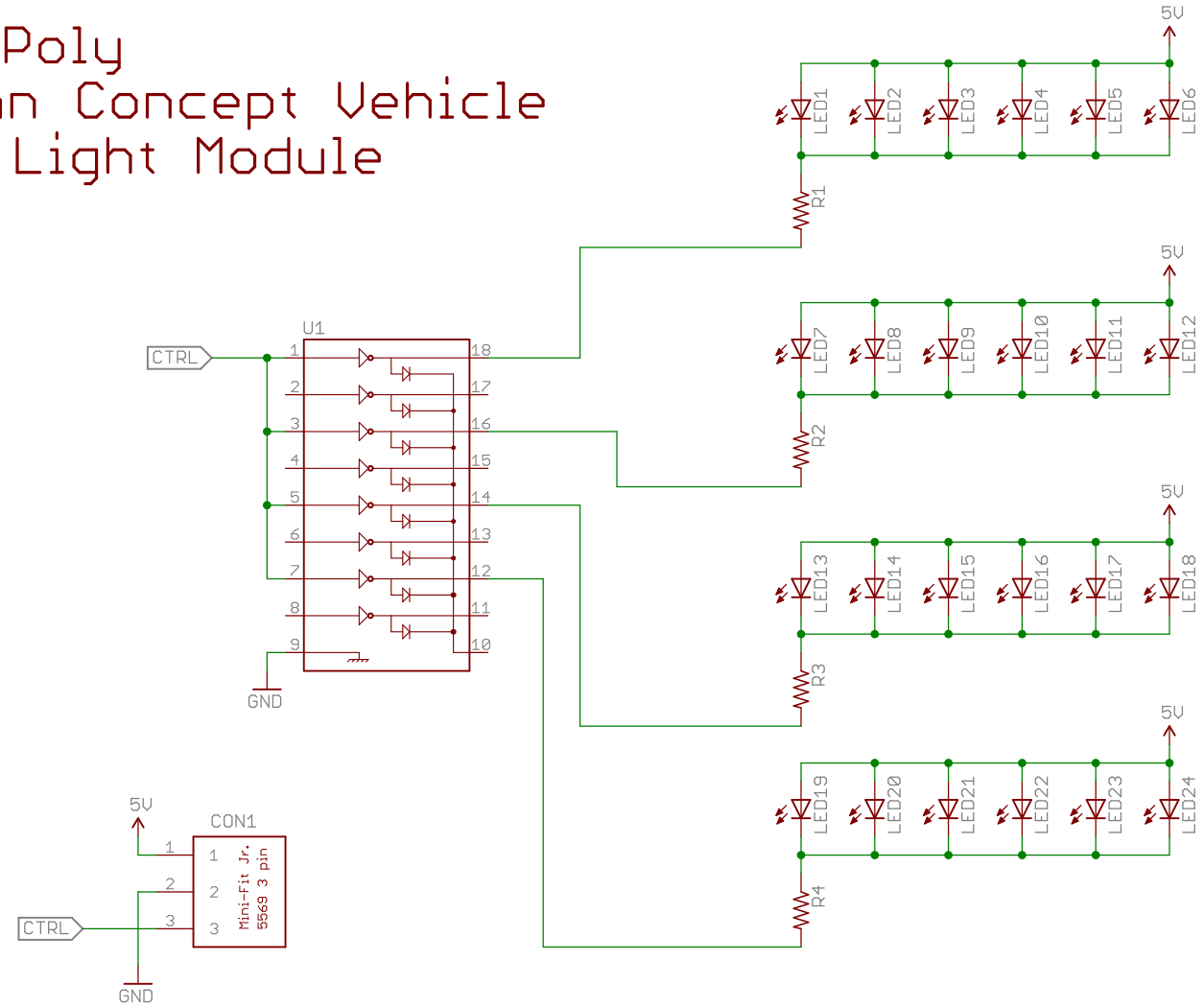
[18] NEC Electronics PS2501-H-A High Isolation Voltage Photocoupler Datasheet. http://www.cel.com/pdf/datasheets/ps2501.pdf.

[19] XBee-PRO 900 RF Module Manual. http://www.sparkfun.com/datasheets/Wireless/Zigbee/XBee-900-Manual.pdf.

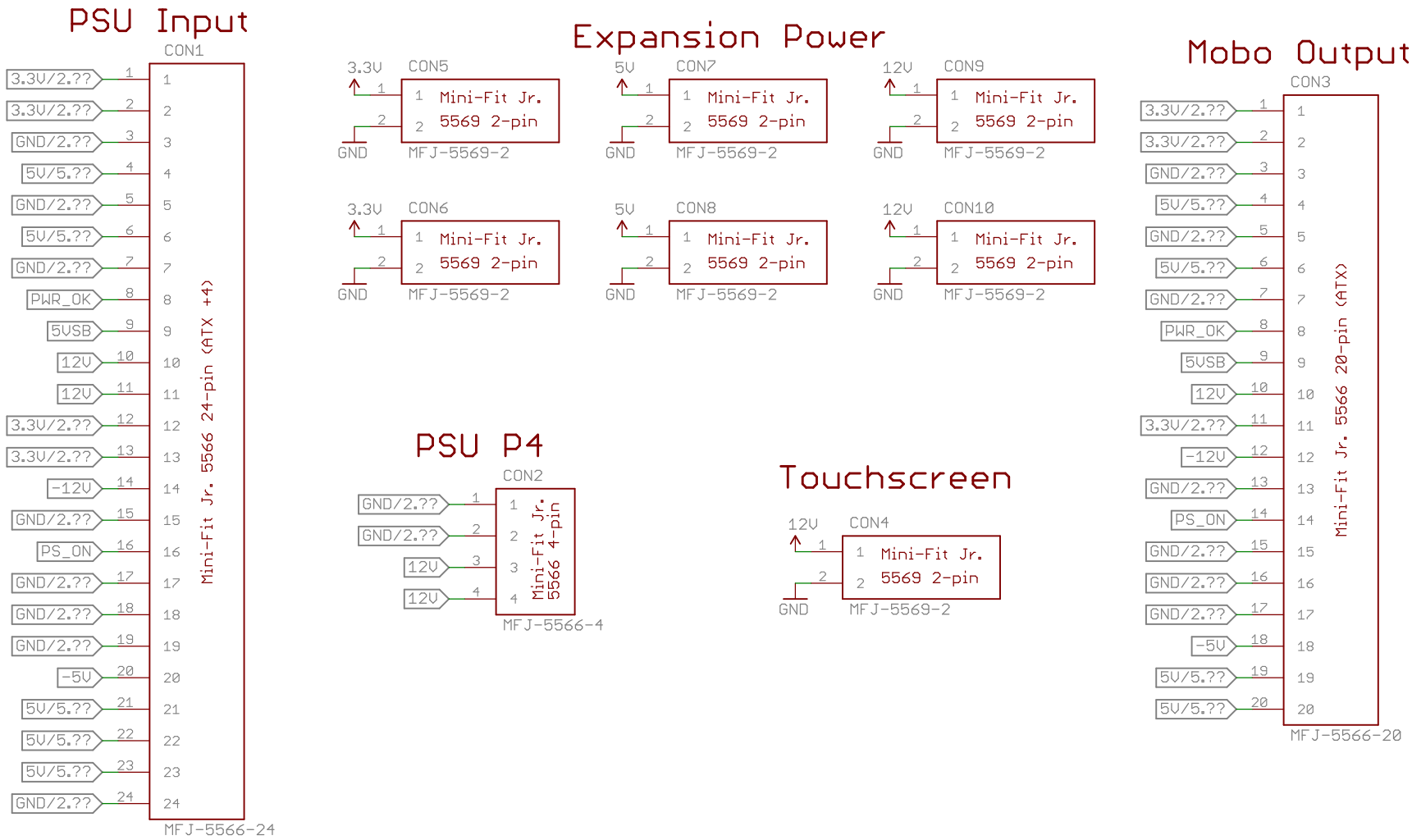
[20] Digi International X-CTU Configuration and Test Software. http://www.digi.com/support/productdetl.jsp?pid=3352&osvid=57&tp=5&s=316.

Appendix A: Schematics

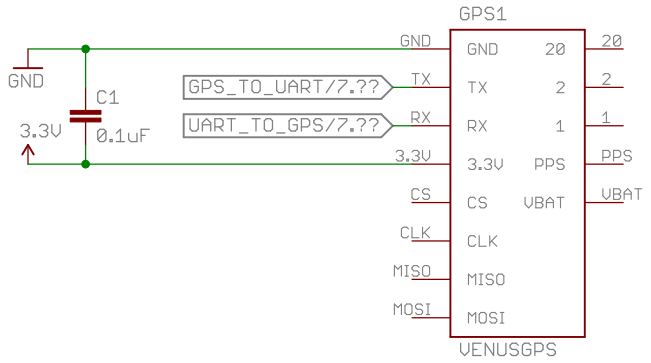
Cal Poly
Urban Concept Vehicle
LED Light Module



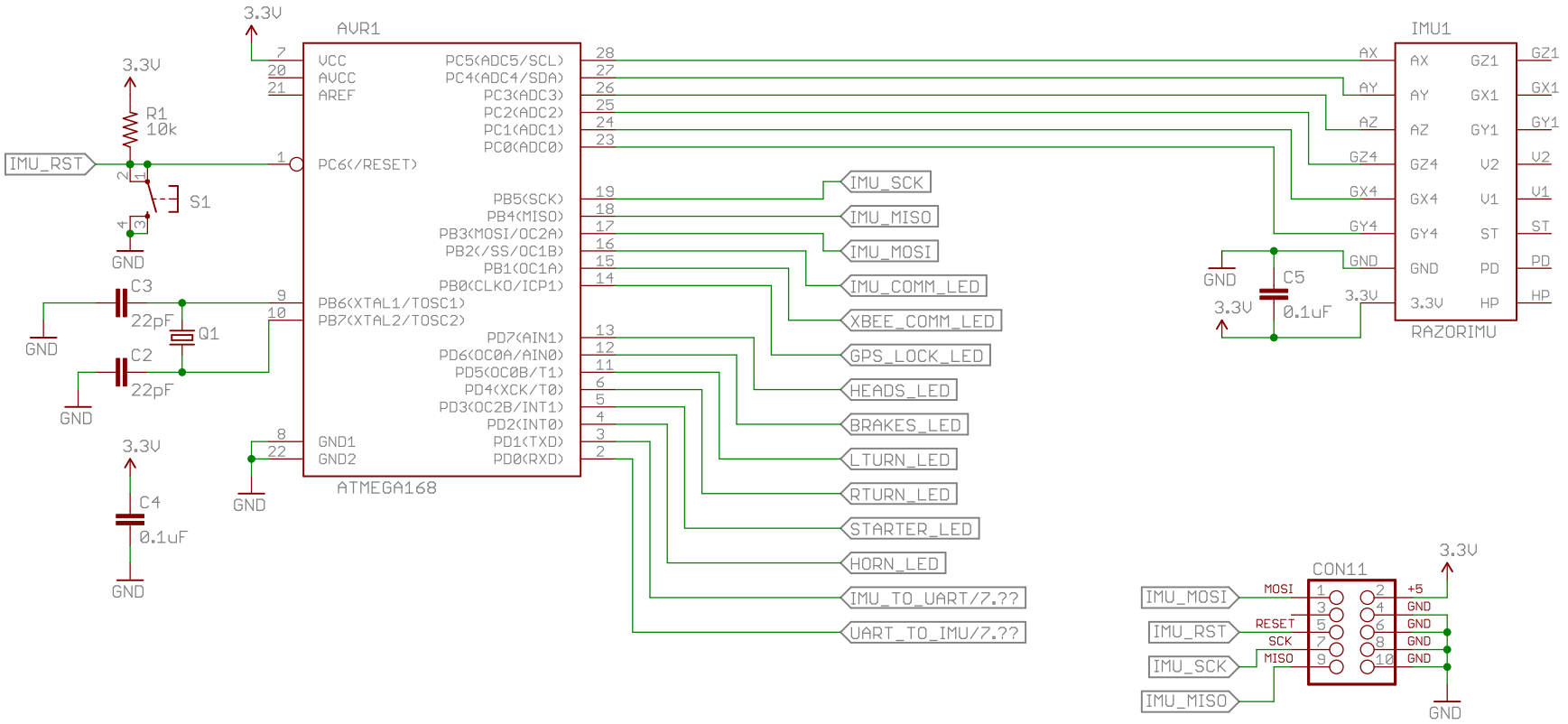
Cal Poly Supermileage Vehicle Team Urban Concept Vehicle	
TITLE: lights	
Document Number: Bob Somers (rsomers@calpoly.edu)	REV: A
Date: 1/28/2010 4:28:28 AM	Sheet: 1/1



Cal Poly Supermileage Vehicle Team Urban Concept: Power Supply	
TITLE: mainboard	
Document Number: Bob Somers (rsomers@calpoly.edu)	REV: A
Date: 3/8/2010 1:29:46 PM	Sheet: 1/8



Cal Poly Supermileage Vehicle Team Urban Concept: GPS		
TITLE: mainboard		
Document Number: Bob Somers (rsomers@calpoly.edu)		REV: A
Date: 3/8/2010 1:29:46 PM	Sheet: 2/8	



Cal Poly Supermileage Vehicle Team
Urban Concept: IMU and Status

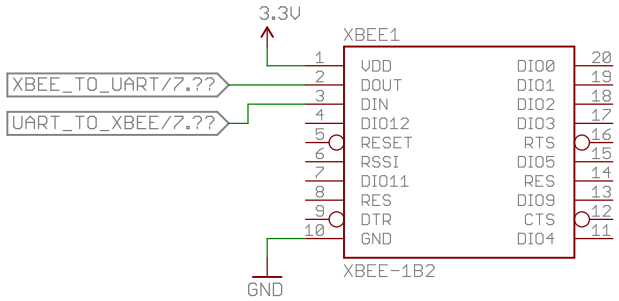
TITLE: mainboard

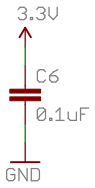
Document Number:
Bob Somers (rsomers@calpoly.edu)

REV: A

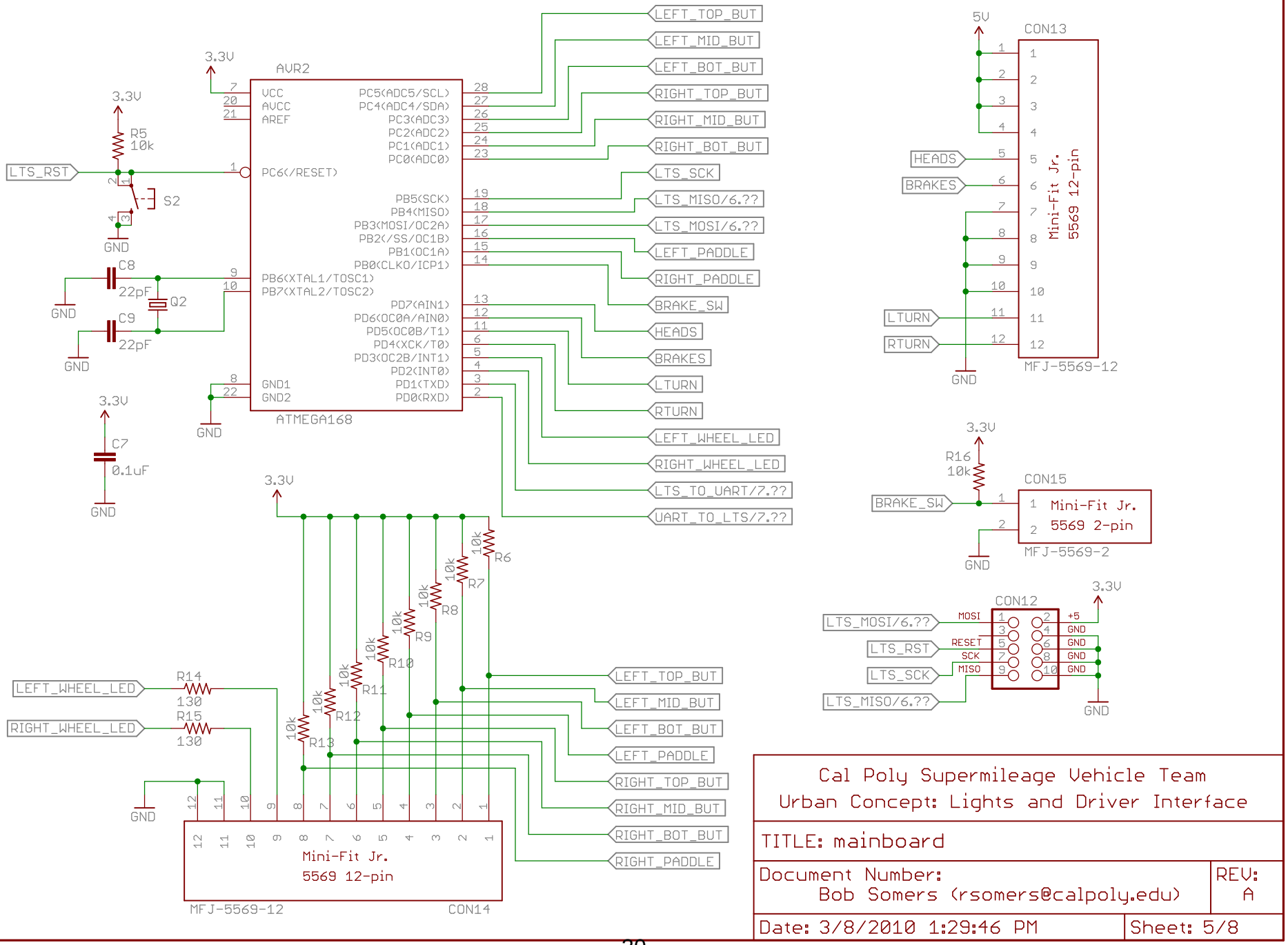
Date: 3/8/2010 1:29:46 PM

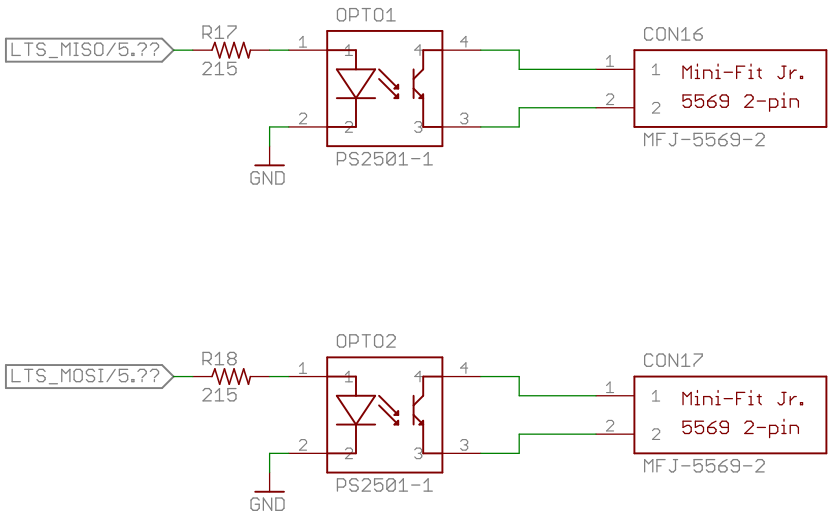
Sheet: 3/8



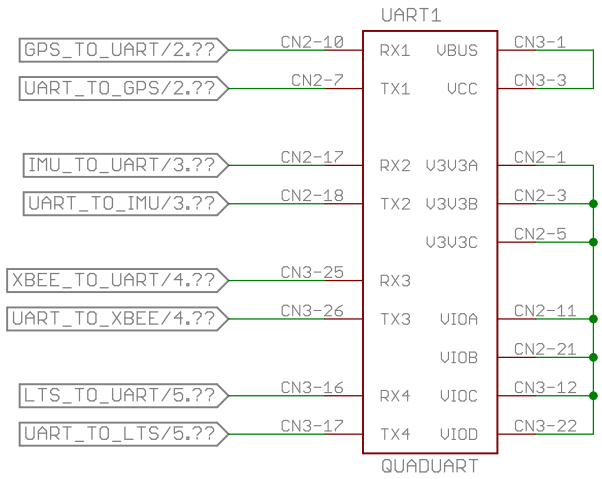


Cal Poly Supermileage Vehicle Team Urban Concept: Wireless	
TITLE: mainboard	
Document Number: Bob Somers (rsomers@calpoly.edu)	REV: A
Date: 3/8/2010 1:29:46 PM	Sheet: 4/8

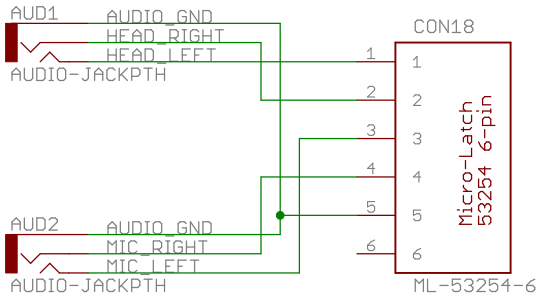

AVR2
ATMEGA168
3.3V
5V
GND
LTS_RST
S2
R5
10k
C8
22pF
C9
22pF
Q2
C7
0.1uF
LEFT_TOP_BUT
LEFT_MID_BUT
LEFT_BOT_BUT
RIGHT_TOP_BUT
RIGHT_MID_BUT
RIGHT_BOT_BUT
LTS_SCK
LTS_MISO/6.??
LTS_MOSI/6.??
LEFT_PADDLE
RIGHT_PADDLE
BRAKE_SW
HEADS
BRAKES
LTURN
RTURN
LEFT_WHEEL_LED
RIGHT_WHEEL_LED
LTS_TO_UART/7.??
UART_TO_LTS/7.??
R6
R7
R8
R9
R10
R11
R12
R13
R14
130
R15
130
CON14
MFJ-5569-12
Mini-Fit Jr.
5569 12-pin
CON13
MFJ-5569-12
CON15
MFJ-5569-2
R16
10k
CON12
MOSI
RESET
SCK
MISO
+5
Cal Poly Supermileage Vehicle Team
Urban Concept: Lights and Driver Interface
TITLE: mainboard
Document Number:
Bob Somers (rsomers@calpoly.edu)
REV: A
Date: 3/8/2010 1:29:46 PM
Sheet: 5/8



Cal Poly Supermileage Vehicle Team Urban Concept: Starter/Horn Driver	
TITLE: mainboard	
Document Number: Bob Somers (rsomers@calpoly.edu)	REV: A
Date: 3/8/2010 1:29:46 PM	Sheet: 6/8

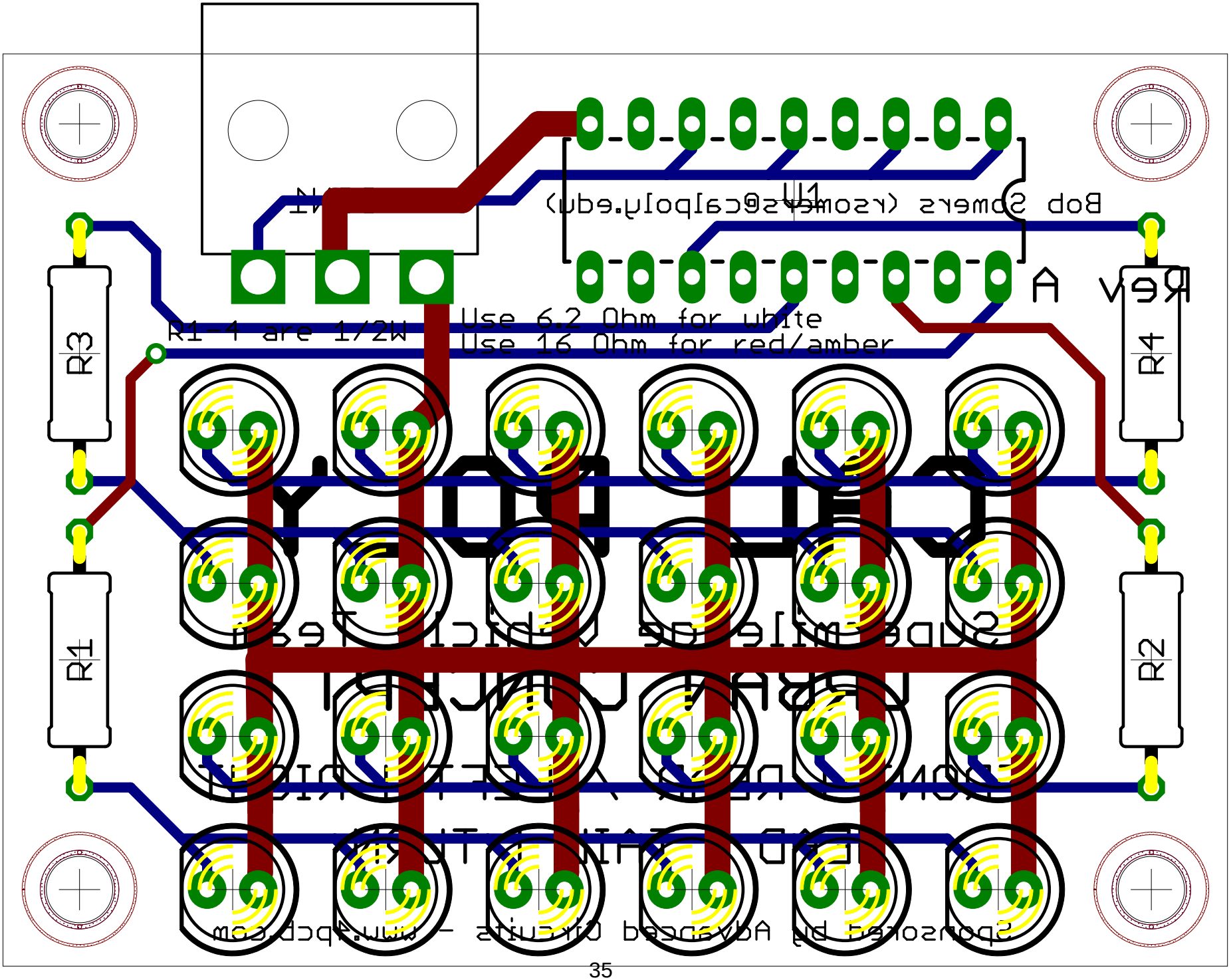


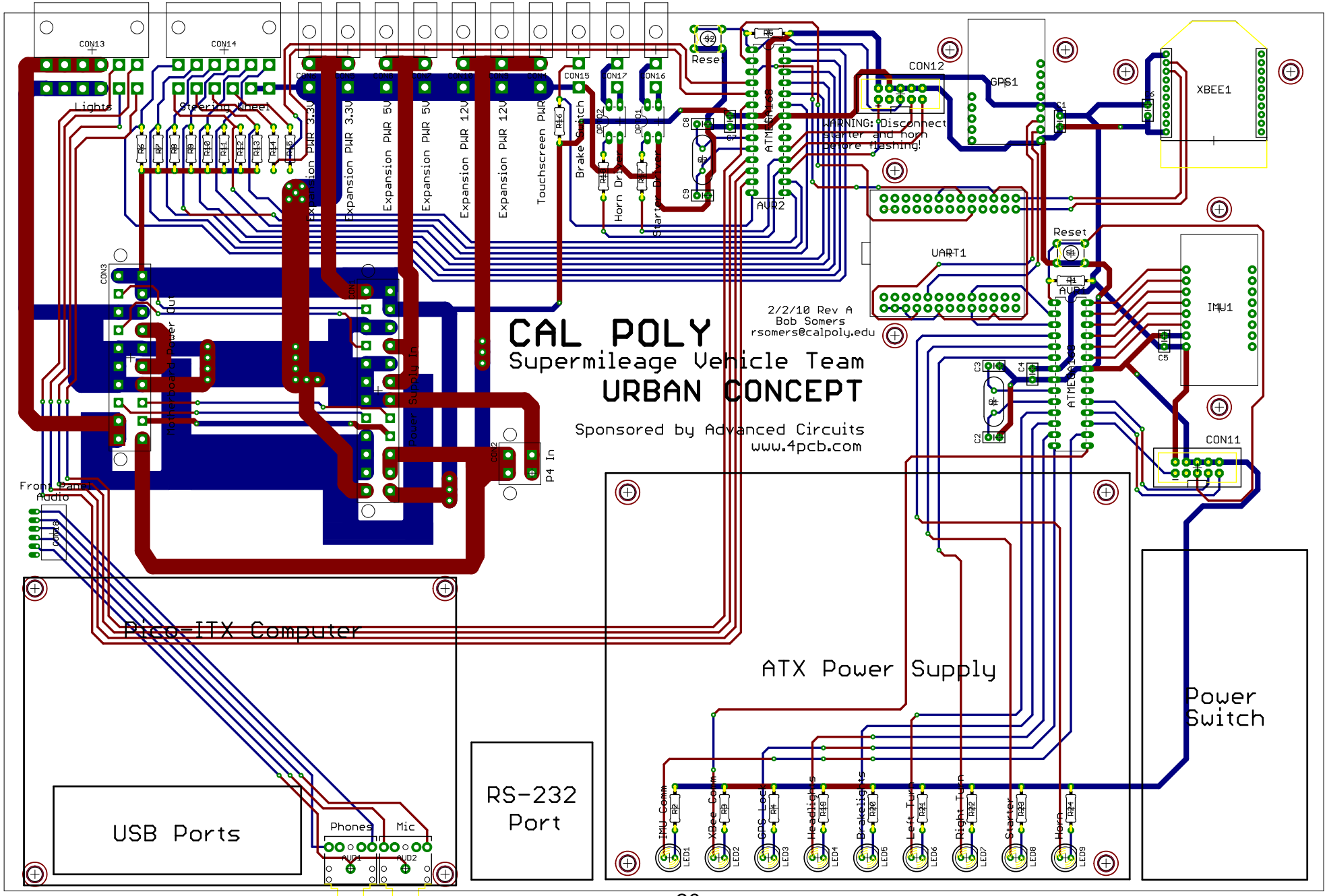
Cal Poly Supermileage Vehicle Team Urban Concept: Quad UART to USB		
TITLE: mainboard		
Document Number: Bob Somers (rsomers@calpoly.edu)		REV: A
Date: 3/8/2010 1:29:46 PM	Sheet: 7/8	



Cal Poly Supermileage Vehicle Team Urban Concept: Front Panel Audio		
TITLE: mainboard		
Document Number: Bob Somers (rsomers@calpoly.edu)		REV: A
Date: 3/8/2010 1:29:46 PM	Sheet: 8/8	

Appendix B: Board Layouts


R1-4 are 1/2W
Use 6.2 Ohm for white
Use 16 Ohm for red/amber
R1
R2
R3
R4
U1
Rev A


CAL POLY
Supermileage Vehicle Team
URBAN CONCEPT
Sponsored by Advanced Circuits
www.4pcb.com
2/2/10 Rev A
Bob Somers
rsomers@calpoly.edu
WARNING: Disconnect starter and horn before flashing!
Pico-ITX Computer
ATX Power Supply
Power Switch
USB Ports
RS-232 Port
Phones
Mic
Reset
CON12
GPS1
XBEE1
UART1
IMU1
CON11
Lights
Steering Wheel
Front Panel Audio
Motherboard Power
Power Supply In
P4 In
Brake Switch
Horn Driver
Starter Driver
Touchscreen PWR
Expansion PWR 3.3V
Expansion PWR 5V
Expansion PWR 12V
IMU Comm
XBee Comm
GPS Loc
Headlights
Brakelights
Left Turn
Right Turn
Starter
Horn

Appendix C: Code Listings

The code running on the two embedded microcontrollers is included in this section. For more extensive source code listings (such as the complete source code for the virtual dashboard, which is much too large to include here), see the Google Code project at:

http://code.google.com/p/cpsmv/

IMU Microcontroller Interface Code

```
// ========================================
// IMU INTERFACE CODE
// Bob Somers, 2010
// Cal Poly Supermileage Vehicle Team
// California Polytechnic State University
// San Luis Obispo, CA
//
// This code is designed to run on an
// ATmega168 microcontroller with our custom
// mainboard. The microcontroller should be
// clocked at 7.3728 MHz.
//
// This handles the interfacing with the analog
// inertial measurement unit (IMU) and taking,
// 10-bit ADC measurements every 1/10th of a
// second. These measurements are transmitted
//  to the host PC over the ATmega168's UART at
// 115200 baud. Status commands can also be sent
// from the host PC to this chip which allows
// control of 8 status LEDs on the mainboard.
// ========================================

#include <stdint.h>
#include <avr/io.h>

#define IMU_COMM 0
#define XBEE_COMM 1
#define GPS_LOCK 2
#define HEAD_LTS 3
#define BRAKE_LTS 4
#define LTURN_LTS 5
#define RTURN_LTS 6
#define STARTER 7
#define HORN 8

// WARNING: several raw register settings make the assumption this
// system is running on a system clock of 7.3728 MHz!

// ========================================
// ADC FUNCTIONS
// ========================================
```

```
void adc_init()
{
        // disable internal pull-ups, since they interfere
        // with the adc
        PORTC = 0x00;
        DDRC = 0x00;

        // take the reference voltage from AVcc, and initially set the mux to sample gnd
        ADMUX = (1 << REFS0) | 0x0f;

        // enable the adc, use a 64 as the clock divider (7.3728 MHz / 64 gives us a
        // 115200 KHz clock, and the adc needs between 50 KHz and 200 KHz), and start a
        // conversion to "prime" the system (the first conversion takes significantly
        // longer than subsequent conversions)
        ADCSRA = (1 << ADEN) | (1 << ADSC) | 0x06;

        // wait for the first (dummy) conversion to complete
        while (ADCSRA & (1 << ADSC));
}
uint16_t adc_sample(uint8_t channel)
{
        uint8_t low, high, mux;

        // set the mux to the selected channel, while preserving the upper 4 bits
        mux = (ADMUX & 0xf0) | (channel & 0x0f);
        ADMUX = mux;

        // start the conversion and wait for it to complete
        ADCSRA |= (1 << ADSC);
        while (ADCSRA & (1 << ADSC));

        // read out the value, LOW BYTE FIRST, so that the data isn't cleared before
        // we finish reading it
        low = ADCL;
        high = ADCH;

        // return the 10-bit sample
        return ((uint16_t)high << 8) | low;
}
// ========================================
// UART FUNCTIONS
// ========================================
void uart_init()
{
        // based on 7.3728 MHz clock and a 115200 bps baud rate
        UBRR0H = 0;
        UBRR0L = 3;

        // set the frame format to 8 data bits, no parity, 1 stop bit
        UCSR0C = (0 << USBS0) | (1 << UCSZ01) | (1 << UCSZ00);

        // enable both the receiver and the transmitter
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
}
void uart_send(uint8_t byte)
{
        // wait until the uart is ready
```

```
        while ((UCSR0A & (1 << UDRE0)) == 0);

        // fire off the data byte
        UDR0 = byte;
}
// asynchronous, returns true/false immediately if data contains a valid byte
uint8_t uart_receive(uint8_t *data)
{
        if (!(UCSR0A & (1 << RXC0)))
        {
                // nothing ready
                return 0;
        }
        // set the data byte
        *data = UDR0;
        // success
        return 1;
}
void uart_print(char *string, int len)
{
        int i;
        for (i = 0; i < len; i++)
        {
                uart_send(string[i]);
        }
}
// ========================================
// LED FUNCTIONS
// ========================================
void leds_init()
{
        // put B2, B1, and B0 in output mode
        DDRB |= 0x7;
        // put D7, D6, D5, D4, D3, D2 in output mode
        DDRD |= (0x3f << 2);
}
void led_set(uint8_t which, uint8_t status)
{
        switch (which)
        {
                case IMU_COMM:
                        if (status)
                        {
                                PORTB |= (1 << 2);
                        }
                        else
                        {
                                PORTB &= ~(1 << 2);
                        }
                        break;
```

```
case XBEE_COMM:
        if (status)
        {
                PORTB |= (1 << 1);
        }
        else
        {
                PORTB &= ~(1 << 1);
        }
        break;
case GPS_LOCK:
        if (status)
        {
                PORTB |= (1 << 0);
        }
        else
        {
                PORTB &= ~(1 << 0);
        }
        break;

case HEAD_LTS:
        if (status)
        {
                PORTD |= (1 << 7);
        }
        else
        {
                PORTD &= ~(1 << 7);
        }
        break;

case BRAKE_LTS:
        if (status)
        {
                PORTD |= (1 << 6);
        }
        else
        {
                PORTD &= ~(1 << 6);
        }
        break;

case LTURN_LTS:
        if (status)
        {
                PORTD |= (1 << 5);
        }
        else
        {
                PORTD &= ~(1 << 5);
```

```
                        }
                        break;
                case RTURN_LTS:
                        if (status)
                        {
                                PORTD |= (1 << 4);
                        }
                        else
                        {
                                PORTD &= ~(1 << 4);
                        }
                        break;
                case STARTER:
                        if (status)
                        {
                                PORTD |= (1 << 3);
                        }
                        else
                        {
                                PORTD &= ~(1 << 3);
                        }
                        break;
                case HORN:
                        if (status)
                        {
                                PORTD |= (1 << 2);
                        }
                        else
                        {
                                PORTD &= ~(1 << 2);
                        }
                        break;
                default:
                        break;
        }
}
// ========================================
// TIMER FUNCTIONS
// ========================================
void timer_init()
{
        // set the prescaler on the 16-bit timer to 64
        TCCR1B |= (1 << CS10) | (1 << CS11);
}
uint16_t timer_get()
{
        // return the current count
        return TCNT1;
}
void timer_reset()
```

```
{
        // reset the count register
        TCNT1 = 0;
}
// ========================================
// ENTRY POINT
// ========================================
int main()
{
        uint16_t adc_value[6];
        uint8_t digit[6][4];
        uint8_t i;
        uint8_t leds[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
        uint8_t imu_comm_ctr = 0;
        uint8_t uart_buffer[3] = {0, 0, 0};
        uint8_t byte;
        // get the subsystems up and running
        uart_init();
        adc_init();
        leds_init();
        timer_init();
        timer_reset();
        while (1)
        {
                // is it time to do a sample (has 1/10th of a second passed?)
                if (timer_get() >= 11520)
                {
                        // yes! do the adc sample and send it off down the uart

                        // sample all 6 channels
                        for (i = 0; i < 6; i++)
                        {
                                adc_value[i] = adc_sample(i);
                        }
                        // convert to digits
                        for (i = 0; i < 6; i++)
                        {
                                digit[i][3] = adc_value[i] / 1000;
                                adc_value[i] = adc_value[i] % 1000;
                                digit[i][2] = adc_value[i] / 100;
                                adc_value[i] = adc_value[i] % 100;
                                digit[i][1] = adc_value[i] / 10;
                                adc_value[i] = adc_value[i] % 10;
                                digit[i][0] = adc_value[i];
                        }

                        // output to the uart
                        uart_send('$'); // start of line character
                        for (i = 0; i < 6; i++)
                        {
                                uart_send(digit[i][3] + 48);
```

```
                uart_send(digit[i][2] + 48);
                uart_send(digit[i][1] + 48);
                uart_send(digit[i][0] + 48);
                uart_send('\t');
        }
        uart_send('\r');
        uart_send('\n');

        // increment the imu comm counter
        imu_comm_ctr++;
        // restart the timer
        timer_reset();
}
// update imu comm led status
if (imu_comm_ctr > 0)
{
        leds[IMU_COMM] = !leds[IMU_COMM];
        led_set(IMU_COMM, leds[IMU_COMM]);
        imu_comm_ctr = 0;
}
// try to get a byte from the uart
if (uart_receive(&byte))
{
        // shift the buffer
        uart_buffer[0] = uart_buffer[1];
        uart_buffer[1] = uart_buffer[2];
        uart_buffer[2] = byte;
        // middle byte MUST be a '=' for any of these to be true
        if (uart_buffer[1] == '=')
        {
                switch (uart_buffer[0])
                {
                        case 'x':
                                led_set(XBEE_COMM, uart_buffer[2] - 48);
                                break;
                        case 'g':
                                led_set(GPS_LOCK, uart_buffer[2] - 48);
                                break;
                        case 'h':
                                led_set(HEAD_LTS, uart_buffer[2] - 48);
                                break;
                        case 'b':
                                led_set(BRAKE_LTS, uart_buffer[2] - 48);
                                break;
                        case 'l':
                                led_set(LTURN_LTS, uart_buffer[2] - 48);
                                break;
                        case 'r':
                                led_set(RTURN_LTS, uart_buffer[2] - 48);
                                break;
```

```
                                        case 's':
                                                led_set(STARTER, uart_buffer[2] - 48);
                                                break;
                                        case 'n':
                                                led_set(HORN, uart_buffer[2] - 48);
                                                break;
                                        default:
                                                break;
                                }
                        }
                }
        }
        return 0;
}
```

Driver Microcontroller Interface Code

```
// ========================================
// DRIVER INTERFACE CODE
// Bob Somers, 2010
// Cal Poly Supermileage Vehicle Team
// California Polytechnic State University
// San Luis Obispo, CA
//
// This code is designed to run on an
// ATmega168 microcontroller with our custom
// mainboard. The microcontroller should be
// clocked at 7.3728 MHz.
//
// This handles the driver interface details,
// such as getting button states from the
// steering wheel, updating the states of the
// lighting systems, starter, and horn, and
// communicating system status to the host PC
// over the ATmega168's UART at 115200 baud.
// ========================================

#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define HEAD_LTS 0
#define BRAKE_LTS 1
#define LTURN_LTS 2
#define RTURN_LTS 3
#define DRIVER_LEFT 4
#define DRIVER_RIGHT 5
#define STARTER 6
#define HORN 7

#define LEFT_TOP 0
#define LEFT_MID 1
```

```
#define LEFT_BOT 2
#define LEFT_PAD 3
#define RIGHT_TOP 4
#define RIGHT_MID 5
#define RIGHT_BOT 6
#define RIGHT_PAD 7
#define BRAKE_SW 8

// global variable for locking out buttons during debounce period
volatile int buttons_locked_out = 0;

// WARNING: several raw register settings make the assumption this
// system is running on a system clock of 7.3728 MHz!

// ========================================
// UART FUNCTIONS
// ========================================

void uart_init()
{
        // based on 7.3728 MHz clock and a 115200 bps baud rate
        UBRR0H = 0;
        UBRR0L = 3;

        // set the frame format to 8 data bits, no parity, 1 stop bit
        UCSR0C = (0 << USBS0) | (1 << UCSZ01) | (1 << UCSZ00);

        // enable both the receiver and the transmitter
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
}

void uart_send(uint8_t byte)
{
        // wait until the uart is ready
        while ((UCSR0A & (1 << UDRE0)) == 0);

        // fire off the data byte
        UDR0 = byte;
}

// asynchronous, returns true/false immediately if data contains a valid byte
uint8_t uart_receive(uint8_t *data)
{
        if (!(UCSR0A & (1 << RXC0)))
        {
                // nothing ready
                return 0;
        }

        // set the data byte
        *data = UDR0;

        // success
        return 1;
}

void uart_print(char *string, int len)
{
        int i;
```

```
        for (i = 0; i < len; i++)
        {
                uart_send(string[i]);
        }
}
// ========================================
// LED FUNCTIONS
// ========================================
void leds_init()
{
        // put B4 and B3 in output mode
        DDRB |= (0x3 << 3);
        // put D7, D6, D5, D4, D3, D2 in output mode (D1 and D0 are uart)
        DDRD |= (0x3f << 2);
}
void led_set(uint8_t which, uint8_t status)
{
        switch (which)
        {
                case STARTER:
                        if (status)
                        {
                                PORTB |= (1 << 4);
                        }
                        else
                        {
                                PORTB &= ~(1 << 4);
                        }
                        break;
                case HORN:
                        if (status)
                        {
                                PORTB |= (1 << 3);
                        }
                        else
                        {
                                PORTB &= ~(1 << 3);
                        }
                        break;
                case HEAD_LTS:
                        if (status)
                        {
                                PORTD |= (1 << 7);
                        }
                        else
                        {
                                PORTD &= ~(1 << 7);
                        }
                        break;
```

```
case BRAKE_LTS:
        if (status)
        {
                PORTD |= (1 << 6);
        }
        else
        {
                PORTD &= ~(1 << 6);
        }
        break;

case LTURN_LTS:
        if (status)
        {
                PORTD |= (1 << 5);
        }
        else
        {
                PORTD &= ~(1 << 5);
        }
        break;

case RTURN_LTS:
        if (status)
        {
                PORTD |= (1 << 4);
        }
        else
        {
                PORTD &= ~(1 << 4);
        }
        break;
case DRIVER_LEFT:
        if (status)
        {
                PORTD |= (1 << 3);
        }
        else
        {
                PORTD &= ~(1 << 3);
        }
        break;
case DRIVER_RIGHT:
        if (status)
        {
                PORTD |= (1 << 2);
        }
        else
        {
                PORTD &= ~(1 << 2);
        }
```

```
                        break;
                default:
                        break;
        }
}
// ========================================
// BUTTON FUNCTIONS
// ========================================
void buttons_init()
{
        // put C5, C4, C3, C2, C1, and C0 in input mode
        DDRC &= ~(0x3f);
        // put B2, B1, and B0 in input mode
        DDRC &= ~(0x7);
}
uint8_t button_get(uint8_t which)
{
        switch (which)
        {
                case LEFT_TOP:
                        return !(PINC & (1 << 5));
                        break;
                case LEFT_MID:
                        return !(PINC & (1 << 4));
                        break;
                case LEFT_BOT:
                        return !(PINC & (1 << 3));
                        break;

                case LEFT_PAD:
                        return !(PINB & (1 << 2));
                        break;

                case RIGHT_TOP:
                        return !(PINC & (1 << 2));
                        break;

                case RIGHT_MID:
                        return !(PINC & (1 << 1));
                        break;
                case RIGHT_BOT:
                        return !(PINC & (1 << 0));
                        break;
                case RIGHT_PAD:
                        return !(PINB & (1 << 1));
                        break;
                case BRAKE_SW:
                        return !(PINB & (1 << 0));
                        break;
```

```
                default:
                        break;
        }
        return 0;
}
// ========================================
// TIMER FUNCTIONS
// ========================================
void timer_init()
{
        // set the prescaler on the 16-bit timer to 64
        TCCR1B |= (1 << CS10) | (1 << CS11);
}
uint16_t timer_get()
{
        // return the current count
        return TCNT1;
}
void timer_reset()
{
        // reset the count register
        TCNT1 = 0;
}
void debouncer_init()
{
        // set the prescaler on the 8-bit timer to 1024
        TCCR0B |= (1 << CS02) | (1 << CS00);
}
void debouncer_lockout()
{
        // don't lock out buttons if they're already locked out
        if (buttons_locked_out) return;

        buttons_locked_out = 1;

        // reset the timer
        TCNT0 = 0;
        // enable overflow interrupts
        TIMSK0 |= (1 << TOIE0);
}
void debouncer_clear()
{
        // don't bother if buttons are already enabled
        if (!buttons_locked_out) return;
        // disable overflow interrupts
        TIMSK0 &= ~(1 << TOIE0);
        buttons_locked_out = 0;
}
ISR(TIMER0_OVF_vect)
```

```
{
        static int count = 0;

        count++;
        if (count > 15)
        {
                count = 0;
                debouncer_clear();
        }
}
// ========================================
// ENTRY POINT
// ========================================
int main()
{
        uint8_t buttons[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
        uint8_t prev[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
        uint8_t leds[8] = {0, 0, 0, 0, 0, 0, 0, 0};
        uint8_t brake_pwm = 0;
        uint8_t turn_pwm = 0;
        uint8_t turn_ctr = 0;
        uint8_t lts_state = 0;
        uint8_t rt_state = 0;
        uint8_t lf_state = 0;
        uint8_t hzd_state = 0;

        // get the subsystems up and running
        uart_init();
        leds_init();
        buttons_init();
        timer_init();
        debouncer_init();

        // enable interrupts globally
        sei();

        timer_reset();
        while (1)
        {
                // save previous button states
                prev[LEFT_TOP] = buttons[LEFT_TOP];
                prev[LEFT_MID] = buttons[LEFT_MID];
                prev[LEFT_BOT] = buttons[LEFT_BOT];
                prev[LEFT_PAD] = buttons[LEFT_PAD];
                prev[RIGHT_TOP] = buttons[RIGHT_TOP];
                prev[RIGHT_MID] = buttons[RIGHT_MID];
                prev[RIGHT_BOT] = buttons[RIGHT_BOT];
                prev[RIGHT_PAD] = buttons[RIGHT_PAD];
                prev[BRAKE_SW] = buttons[BRAKE_SW];

                // get input from the buttons
                buttons[LEFT_TOP] = button_get(LEFT_TOP);
                buttons[LEFT_MID] = button_get(LEFT_MID);
                buttons[LEFT_BOT] = button_get(LEFT_BOT);
                buttons[LEFT_PAD] = button_get(LEFT_PAD);
```

```
buttons[RIGHT_TOP] = button_get(RIGHT_TOP);
buttons[RIGHT_MID] = button_get(RIGHT_MID);
buttons[RIGHT_BOT] = button_get(RIGHT_BOT);
buttons[RIGHT_PAD] = button_get(RIGHT_PAD);
buttons[BRAKE_SW] = button_get(BRAKE_SW);

// generate pwm signals
if (timer_get() < 100)
{
        brake_pwm = 1;
}
else if (timer_get() > 1000)
{
        turn_ctr++;
        timer_reset();
}
else
{
        brake_pwm = 0;
}
if (turn_ctr > 50)
{
        turn_pwm = !turn_pwm;
        turn_ctr = 0;
}

// pass through starter and horn directly
leds[STARTER] = button_get(RIGHT_TOP);
leds[HORN] = button_get(LEFT_TOP);

// if the previous RIGHT_MID state was high and now it's low, falling edge!
if (prev[RIGHT_MID] && !buttons[RIGHT_MID] && !buttons_locked_out)
{
        lts_state = !lts_state; // toggle
        debouncer_lockout();
        uart_send('$'); uart_send('h'); uart_send('=');
        uart_send(lts_state + 48); uart_send('\r'); uart_send('\n');
}

// if the previous LEFT_MID state was high and now it's low, falling edge!
if (prev[LEFT_MID] && !buttons[LEFT_MID] && !buttons_locked_out)
{
        hzd_state = !hzd_state; // toggle
        debouncer_lockout();
        uart_send('$'); uart_send('z'); uart_send('=');
        uart_send(hzd_state + 48); uart_send('\r'); uart_send('\n');

        // start with light on
        turn_pwm = 1;
        turn_ctr = 0;
}

// if the previous LEFT_PAD state was high and now it's low, falling edge!
if (prev[LEFT_PAD] && !buttons[LEFT_PAD] && !buttons_locked_out)
{
        lf_state = !lf_state; // toggle
        debouncer_lockout();
```

```
        uart_send('$'); uart_send('l'); uart_send('=');
        uart_send(lf_state + 48); uart_send('\r'); uart_send('\n');

        // start with light on
        turn_pwm = 1;
        turn_ctr = 0;
}

// if the previous RIGHT_PAD state was high and now it's low, falling edge!
if (prev[RIGHT_PAD] && !buttons[RIGHT_PAD] && !buttons_locked_out)
{
        rt_state = !rt_state; // toggle
        debouncer_lockout();
        uart_send('$'); uart_send('r'); uart_send('=');
        uart_send(rt_state + 48); uart_send('\r'); uart_send('\n');

        // start with light on
        turn_pwm = 1;
        turn_ctr = 0;
}

// output uart on every edge change of the brake switch
if (prev[BRAKE_SW] != buttons[BRAKE_SW])
{
        // brake switch is not debounced
        uart_send('$'); uart_send('k'); uart_send('=');
        uart_send(buttons[BRAKE_SW] + 48); uart_send('\r'); uart_send('\n');
}

// output uart on every edge change of the starter
if (prev[RIGHT_TOP] != buttons[RIGHT_TOP])
{
        // starter is not debounced
        uart_send('$'); uart_send('s'); uart_send('=');
        uart_send(buttons[RIGHT_TOP] + 48); uart_send('\r'); uart_send('\n');
}

// output uart on every edge change of the horn
if (prev[LEFT_TOP] != buttons[LEFT_TOP])
{
        // brake switch is not debounced
        uart_send('$'); uart_send('n'); uart_send('=');
        uart_send(buttons[LEFT_TOP] + 48); uart_send('\r'); uart_send('\n');
}

// output uart on every edge change of a
if (prev[LEFT_BOT] != buttons[LEFT_BOT])
{
        uart_send('$'); uart_send('a'); uart_send('=');
        uart_send(buttons[LEFT_BOT] + 48); uart_send('\r'); uart_send('\n');
}

// output uart on every edge change of b
if (prev[RIGHT_BOT] != buttons[RIGHT_BOT])
{
        uart_send('$'); uart_send('b'); uart_send('=');
        uart_send(buttons[RIGHT_BOT] + 48); uart_send('\r'); uart_send('\n');
}

// handle headlights and tail/brake lights
```

```
if (lts_state)
{
        leds[HEAD_LTS] = 1;
        leds[BRAKE_LTS] = (buttons[BRAKE_SW]) ? 1 : brake_pwm;
        leds[DRIVER_RIGHT] = 1;
        leds[DRIVER_LEFT] = buttons[BRAKE_SW];
}
else
{
        leds[HEAD_LTS] = 0;
        leds[BRAKE_LTS] = buttons[BRAKE_SW];
        leds[DRIVER_RIGHT] = 0;
        leds[DRIVER_LEFT] = buttons[BRAKE_SW];
}
// if hazards are engaged, left and right turn are synched
if (hzd_state)
{
        leds[LTURN_LTS] = turn_pwm;
        leds[DRIVER_LEFT] = turn_pwm;
        leds[RTURN_LTS] = turn_pwm;
        leds[DRIVER_RIGHT] = turn_pwm;
}
else
{
        // if not, they operate independently based on their status
        if (lf_state)
        {
                leds[LTURN_LTS] = turn_pwm;
                leds[DRIVER_LEFT] = turn_pwm;
        }
        else
        {
                leds[LTURN_LTS] = 0;
        }
        if (rt_state)
        {
                leds[RTURN_LTS] = turn_pwm;
                leds[DRIVER_RIGHT] = turn_pwm;
        }
        else
        {
                leds[RTURN_LTS] = 0;
        }
}
// finally set the led outputs
led_set(HEAD_LTS, leds[HEAD_LTS]);
led_set(BRAKE_LTS, leds[BRAKE_LTS]);
led_set(LTURN_LTS, leds[LTURN_LTS]);
led_set(RTURN_LTS, leds[RTURN_LTS]);
led_set(DRIVER_LEFT, leds[DRIVER_LEFT]);
led_set(DRIVER_RIGHT, leds[DRIVER_RIGHT]);
```

```
                led_set(STARTER, leds[STARTER]);
                led_set(HORN, leds[HORN]);
        }
        return 0;
}
```